Memristive Memory Processing Unit (MPU) Controller for In-Memory Processing

Rotem Ben Hur
Andrew & Erna Viterbi Faculty of Electrical Engineering
Technion – Israel Institute of Technology
Haifa, Israel 3200003

Shahar Kvatinsky
Andrew & Erna Viterbi Faculty of Electrical Engineering
Technion – Israel Institute of Technology
Haifa, Israel 3200003

***Abstract* – Modern computers suffer from a growing disparity of speed between processor and memory which significantly limits their performance. Additionally, as the number of transistors per chip continues to increase, the operating frequency stabilizes due to the power considerations. One of the leading solutions to these issues is to reduce data transfer by adding processing capabilities into the memory itself. For data-intensive applications, this means a significant improvement in processing capabilities by saving a significant amount of time and energy. Although all the attempts to implement this solution so far were unsuccessful, emerging non-volatile resistive memory technologies (namely, memristors) offer an opportunity for developing a Memory Processing Unit (MPU) based on a technique called 'stateful logic'. The MPU allows adding processing capabilities to the memristive memory cells, thus enabling novel non-von Neumann architectures. The processing within the MPU relies on a sequence of logical operations. This paper presents the design of an MPU controller for executing in-memory computation. Different design techniques to execute processing and storing data within the MPU are described. The MPU controller has been designed and implemented in a VHDL environment and used to execute different operations within the MPU.**

***Keywords* – memristor, memristive systems, logic, MAGIC, MPU, von Neumann architecture, memory controller.**

I. Introduction

Conventional computers are based on von Neumann architecture, where processing and storing of the data are performed by different units (namely, CPU and memory). Over the last few decades, the performance of processors has improved in a much higher pace than this of memories, which has led to today several orders of magnitude performance gap between processor and memory. This gap causes a bottleneck for transferring data between memory and processor, which is usually called the *memory wall*. A way to overcome this bottleneck is by reducing data transfer necessity. One of the ways to achieve this is by performing some of the computations within the memory. Such novel non-von Neumann architecture shall reduce the energy consumption and improve the performance.

Attempts for reducing the memory wall problem by getting the memory closer to the processing unit have been tried before. One of the leading trials is processing near memory (PNM) architecture. In PNM, processing units, on which part of the program is executed, are added to the off-chip DRAM memory and a part of the program is executed in the off-chip memory by a co-processor. One well-known implementation of PNM is Berkeley's Intelligent RAM (IRAM) project [1]. The IRAM project had not become commercially successful, mainly due to the bad integration between DRAM and logic technologies.

To truly overcome this problem, processing within memory using the same cells for both memory and logic is desired. Such integration can be performed using novel emerging nonvolatile memory technologies. These emerging technologies include RRAM, PCM, STT MRAM and others (for simplicity, we refer to all of them as *memristors*). A memristor is a two-port passive element with varying resistance, which changes according to the voltage applied to the device. Due to their speed, low power, scalability, and high endurance [2], memristors that store data as resistance values, are considered as attractive candidates to replace conventional memory technologies (*e.g.*, DRAM and Flash).

Furthermore, memristive technologies have also been explored for additional applications such as logic circuits, whereas some of the proposed logic may be computed within a memristive memory structure [3-5], allowing both storage and processing within the same cells and without changing the topology of the memristive memory array. Such a *Memory Processing Unit* (MPU) that can be dynamically changed from data processing to storage is proposed in [6].

In conventional systems, a memory controller which is responsible for the interface with the memory resides within the processor. Inside the standard memory, located a controller which receives the read and write requests from the memory controller and executes them. The structure of such systems is described in Figure 1a. In our proposed non-von Neumann architecture, an MPU uses as memory. A memory controller is still located within the processor, however, instead of the simple controller within the memory, a more intricate MPU controller is located within the MPU, as described in Figure 1b. In this paper, a description of the MPU controller and its complexity are presented.

II. Logic within Memristive Memories

A. Memristive Memories

Memristive devices are made of a dielectric material which is fabricated between two metal electrodes. The logical state of the memristor is represented as a resistance, where high resistance is considered as logical zero and low resistances is considered as logical one.

Memristive memories enable an extremely dense memory of $4F^2$ (where F is the feature size). The memristive memory structure can either be a crossbar array [7] or include selectors in each memory cell [8]. Thus, undesired current sneak paths in crossbar arrays, where unselected cells interfere with the

978-1-5090-2152-9/16/$31.00 ©2016 IEEE

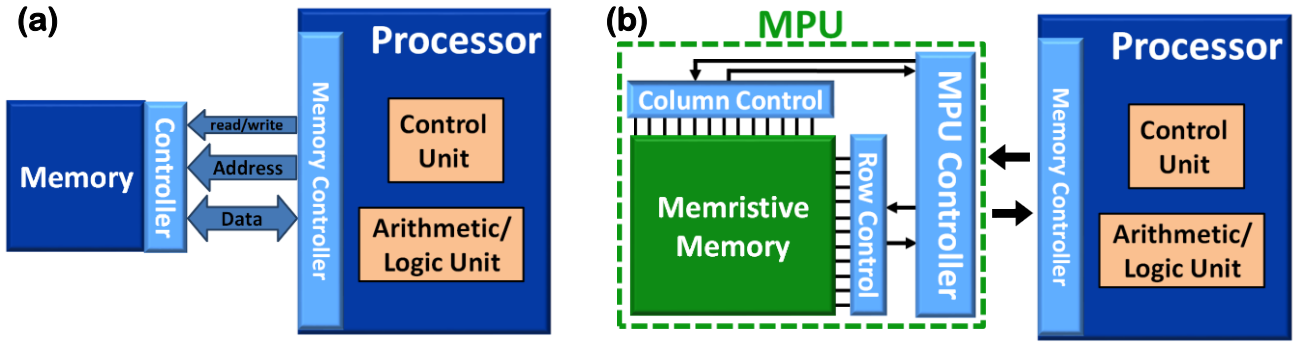

Figure 1. (a) General structure of (a) von-Neumann architectures, and (b) the proposed architecture with an MPU as the memory.

selected cells, are neglected in this paper.

Figure 2 shows the schematic of a memristive crossbar. Writing logical '0' and '1' to a memristor is performed by applying V_{RESET} and V_{SET}, respectively, across it. Read operation is achieved by applying a voltage below threshold V_{READ} across the selected memristor and measuring the current which flows through it using a sense amplifier.

The crossbar structure is symmetrical, thus the operations which are performed in the memory may be executed by applying voltages from both horizontal and vertical directions, using a *transpose memory* [9]. Such memory can be used to access the array from different directions, thus operations on both row and column vectors can be performed.

B. Stateful Logic

In addition to conventional storage capabilities of memristive memories, several logical families which are performed within the memory array have been proposed [10-11]. One of the techniques for such in-memory logic is called *stateful logic*, where the logical state of the memristor is represented solely by the resistance. The inputs and outputs of the logic gates are, respectively, the states of the memristors before and at the end of the computation. The applied voltages across the memristors write the result to the output memristor based on the initially stored values in the input memristor. A few stateful logic families are proposed in [12-15], based on different voltages which are applied across the bitlines and wordlines of the memory array.

C. MAGIC – A Stateful Logic Family

An improved stateful logic family is Memristor-Aided Logic (MAGIC) [5], where only a single voltage V_G is used to perform a NOR logic operation. Since NOR is a complete logic function, MAGIC NOR operation is sufficient to execute any Boolean operation. This NOR gate is the basis for performing all processing within memory, thus each processing task is divided into a sequence of MAGIC NOR operations, which will be executed one after the other using the memory cells as computation elements. The schematic of a MAGIC gate operation, performed over row vectors within a memristive memory is shown in Figure 2.

III. Memory Processing Unit (MPU)

Combining storage with processing capabilities within memories enables the development of novel non-von Neumann

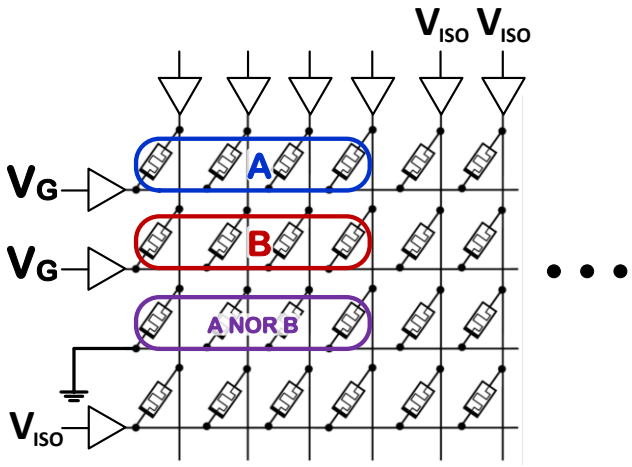

Figure 2. MAGIC NOR operation between two row vectors performed within the memristive memory by applying V_G to the input memristors, ground to the output memristors, and V_{ISO} to unselected bitlines and wordlines. The operation takes one clock cycle for any number of pairs of A_i and B_i [9].

architectures. Although conventional memory technologies (*e.g.*, DRAM and Flash) cannot be used for such a purpose, most memristive memories cells can also act as processing elements. Such a unit which combines memory and computations is a recently proposed Memory Processing Unit (MPU) [6], which consists of a memristive memory and CMOS control, as illustrated in Figure 3. The MPU can either act as an integrated unit, where the memory has independent processing capabilities, or as conventional memory in a von Neumann machine. Therefore, the MPU is compatible with standard computing systems.

To complete the compatibility with conventional von Neumann architecture, standard instruction set architectures (ISA), such as ARM and X86, have to be extended with dedicated in-memory logic and arithmetic instructions. These instructions will be used to perform computation tasks on known locations in the memory (*i.e.*, addresses). This adaptation must also include the development of a compiler which will be able to compile high-level code (*e.g.*, C code) into the extended ISA. Such a compiler should divide the execution of the program between the processor and the MPU. The parts of the program that benefit from execution within the memory shall be written in the extended ISA, while other portions will be executed in conventional von Neumann style.

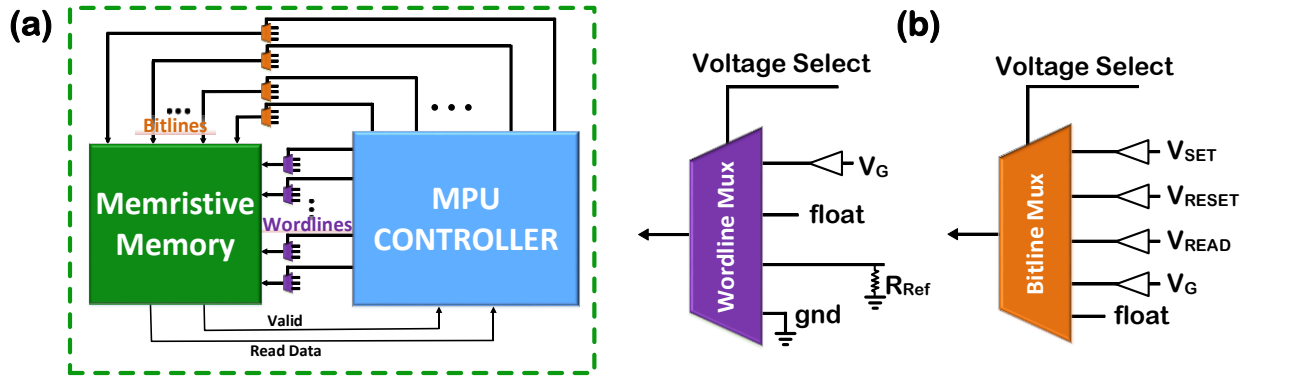

Figure 3. (a) General structure of an MPU. Voltages which are applied to each bitline and wordline are determined using analog multiplexers with a digital select input, which is received from the MPU controller. (b) Description of the wordline and bitline analog muxes.

Since vector operations benefit most from stateful logic, most of the extended ISA are actually vector instructions.

To execute an instruction within the MPU, the memory controller sends the MPU commands along with conventional read and write commands to the MPU, using an extended conventional processor-memory interface protocol (*e.g.*, DDR4 protocol). The command is received by the MPU controller, which interprets it, convert the command into a sequence of MAGIC operations, and then sends the corresponding control signals in order to execute the operations within the memristive memory. These control signals are converted into applied voltages across all wordlines and bitlines of the memristive memory by using single analog multiplexers with digital select inputs for all wordlines and bitlines, as shown in Figure 3. Based on the applied voltages, the resistance of the output memristors may change, thus the logical state of the memory cells is updated, according to the result of the computation. As described in Section 2, the data within the memory acts as the input of the logical operations and the result is immediately stored within the memory cells, without the need to transfer data out of the memory array. As a result, the use of MPU allows alleviating the memory wall, and reducing the system energy.

IV. MPU Controller

A. General Description and Work Principle

MPU controller is responsible for performing the required operations within the memory. The processor sends to the MPU controller standard read and write instructions as well as processing commands. Based on the message from the processor, the controller dynamically decides whether an element is a data storage element or a processing element. To execute a conventional read or write operation, the controller sends the suitable control signals to all bitlines and wordlines of the addressed memristors (depending on the addresses received from the processor). Using analog multiplexers, the control signals determine which voltage(s) to apply (*e.g.*, V_{SET}, V_{RESET}, V_{READ}, as described in Section II-A) to each bitline and wordline.

Performing processing tasks is more complicated since it requires a sequence of logical steps. The MPU controller receives macro-instructions from the processor, and breaks them into numerous micro-instructions. These micro-instructions are built on several levels of abstraction, where the lowest level is the basic logical operation (*i.e.*, MAGIC NOR operation). The MPU controller pipelines the control signals to the memory, changing the applied voltages on each memory clock cycle. Pipelining the micro-instructions maximizes the processing efficiency in terms of speed and energy.

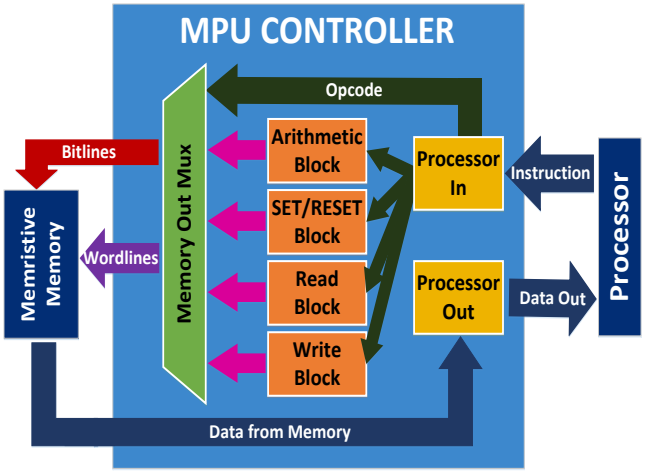

Figure 4. MPU controller block diagram.

We have designed such an MPU controller in VHDL environment. The general structure of the MPU controller is illustrated in Figure 4. An instruction which is sent to the MPU controller enters the 'Processor In' block which interprets it, and sends it for further interpretation to the suitable block, depending on the command type. The available block types are: read, write, SET/RESET and arithmetic. Read and write commands are similar to conventional memory commands. SET and RESET commands writes, respectively, ones or zeroes to areas with dynamic sizes. The MPU controller uses these commands to clear areas of the memory during different stages of the computation processes, or while initializing the memory. While these three block types perform the operation by applying voltages during a single clock cycle, the arithmetic block requires multiple clock cycles to execute different logic and arithmetic operations. The exact number of steps depends on the operation type and vector size. In each step (clock cycle) of the computation, different voltages are applied to the bitlines and wordlines, executing NOR operations in different locations. The arithmetic block is discussed in details in the

next sub-section. Finally, after interpretation in the suitable block, control signals are sent to the memory array through the analog multiplexers and the 'Memory Out Mux' block. In read operations, the read data is sent to the processor by the 'Processor Out' block.

In our design, the latency of all instructions except arithmetic instructions is five clock cycles (according to the stages of the pipeline). Pipelining maintains the maximal throughput of one instruction per cycle. In arithmetic operations, the latency and throughput depends on the number of steps require to execute the logic or arithmetic operation, as described in the next subsection.

B. Arithmetic Block

The arithmetic block is a sophisticated finite state machine, which is responsible for executing the logic and arithmetic commands within the memory. The arithmetic block splits the macro-instructions of arithmetic and logic commands into micro-instructions, which are a set of MAGIC NOR operations.

The arithmetic block consists of a sub-block for each logic or arithmetic command. Each sub-block has NOR and SET blocks. These inner blocks are asynchronous to minimize the stages of the pipeline.

While executing logic and arithmetic operations, the control signals are changed every clock cycle. The throughput, however, is not a single operation per cycle, but determined according to the size of the vectors and the type of logic or arithmetic operation and can be therefore much higher.

C. In-Memory Processing Considerations

To optimize the throughput of the arithmetic execution, different considerations should be taken into account:

1) Algorithms for In-Memory Processing

Different algorithms for executing logic operations which are compatible for in-memory execution (*e.g.*, algorithms based solely on MAGIC NOR operations) needs to be developed. These algorithms maximize the efficiency of the MPU [16]. Exploiting the parallelism offered by the memristive memory is essential to optimize these algorithms in terms of energy, performance, and area. For example, multiplying simultaneously K binary matrices, each of which of size MxN, requires $5NK$-$5K$+$2M$+1 steps when optimizing an algorithm based on MAGIC NOR. This algorithm has a quadratic time complexity of O(NK), while in standard von-Neumann architecture a cubic time complexity of O(NKM) is required. This instance exemplifies the potential performance benefits offered by processing data within the memory. Choosing the right algorithms to process large amount of data can even reach several orders of magnitude improvement.

In the future, automatic design tools, similar to logic synthesis tools, should be evolved to develop new algorithms, while considering the tradeoff between the competency of the algorithm and the complexity of the controller.

2) Processing Area

Executing these algorithms within the MPU requires allocation of dedicated memory cells for computation. The allocation procedure must consider data that is stored within the memory and needs to be retain. Processing therefore may be performed anywhere, as long as the memory cells are not used for storage at the time. In order to keep track of available cells, a dedicated hardware for allocation memory is required. To simplify the allocation mechanism, we allocate memory cells that are dedicated only for processing, as illustrated in Figure 5. Thus before performing a computation, resetting the allocated cells is performed without destructing stored data.

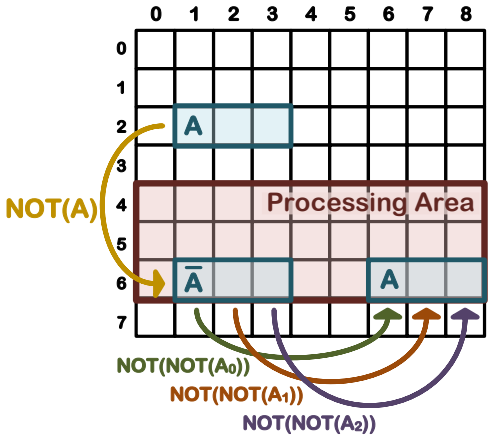

Figure 5. Memory divided into storage and processing areas. Row vector *A* is copied from locations (2,1:3) into the processing area (6,6:8). Since the source address of vector *A* does not share columns with the destination address, then both vector-wise operations as well as bit-wise operations are required. Copying of *A* is performed by NOT(NOT(*A*)), whereas NOT operation is a NOR operation with a single input. The latency of the copy operation is *N*+1 for *N*-bit *A*.

The allocated area for processing and its exact size can be dynamically chosen. Choosing specific locations may reduce energy consumption. For example, by performing the processing using memory cells which are close to the stored data (*e.g.* in matrices among the rows of the matrix). Additionally, changing the location of the processing area can be used as a wear leveling technique [17] to increase the memory lifespan.

3) Vector Operations

Since MAGIC NOR can be executed simultaneously on multiple rows or columns, the MPU can be used to perform vector operations. To effectively perform vector operations, the data needs to be aligned. *i.e.*, when all vector elements laid in the same row (row vector), each element in the input vectors needs to be on the same column to its corresponding element in the other input vector. This alignment enables performing of simultaneous MAGIC NOR operations on all elements in the vector and the performance of the execution of vector operations can be therefore independent on the vector size. If two row vectors rely on the same row, only bit-wise operations can be performed and the performance depends on the vector size. This concept is demonstrated in Figure 5, where NOT(A) is performed in a single parallel operation, while NOT(NOT(A) is performed sequentially in a bit-wise manner. Figure 6 shows simulation results of performing OR logic operation between two row vectors within memory, including the need to move data to the dedicated processing area. Although in some cases bit-wise operations are inevitable, the benefits from vector operations can be significant.

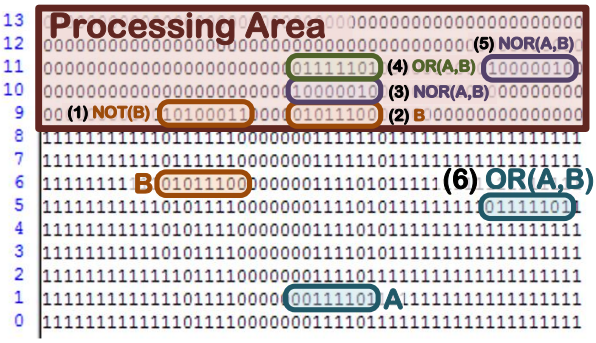

Figure 6. Simulation results of OR operation between two 8-bit vectors *A* and *B* within the processing area. The result is copied to the desired address. The operation is performed by the following sequence of NOR operations:

Copying *B* to align it with *A*:

(1) NOT(*B*) - vector operation (one clock cycle).

(2) NOT(NOT(*B*)) = *B* - bit-wise operations (*N*=8 clock cycles).

Performing the computation between *A* and *B*:

(3) NOR(*A*, *B*) - vector operation (one clock cycle).

(4) NOT(NOR(*A*, *B*)) = OR(*A*, *B*) - vector operation (one clock cycle).

Copying the result to the desired address:

(5) NOT(OR(*A*,*B*))=NOR(*A*,*B*) - bit-wise operation (*N*=8 clock cycles).

(6) NOT(NOR(*A*,*B*)=OR(*A*,*B*) - vector operation (one clock cycle).

Total latency of 2*N*+4=20 clock cycles.

V. Conclusions

This paper discusses the different issues of designing an MPU controller, which is a crucial element for in-memory processing. Processing data in an MPU has a great potential for orders of magnitude improvement in both performance and energy as compared to conventional von Neumann machines.

We have designed an MPU controller in VHDL environment, including developing different algorithms which exploit the parallelism capabilities of the memristive memories and improve the MPU efficiency. In the development of these algorithms, different issues have been considered such as area allocation, and the complexities of the algorithms and the controller.

Acknowledgment

This research was partially supported by Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) and by the Viterbi Fellowship in the Technion Computer Engineering Center.

References

[1] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent DRAM: IRAM," *IEEE Micro*, Vol. 17, No. 2, pp. 34-44, March/April 1997.

[2] S. Kvatinsky E. G. Friedman, A. Kolodny, and U. C. Weiser, "The Desired Memristor for Circuit Designers," *IEEE Circuits and Systems Magazine*, Vol. 13, No. 2, pp. 17-22, Second Quarter 2013.

[3] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based IMPLY Logic Design Flow," *Proceedings of the IEEE International Conference on Computer Design*, pp.142-147, October 2011.

[4] S. Kvatinsky, N. Wald, G. Satat, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI)*, Vol. 22, No. 10, pp. 2054-2066, October 2014.

[5] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC – Memristor Aided LoGIC," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 61, No. 11, pp. 895-899, November 2014.

[6] R. Ben Hur and S. Kvatinsky, "Memory Processing Unit for In-Memory Processing," *Proceedings of the IEEE International Symposium on Nanoscale Architectures*, July 2016.

[7] Z. Jiang, P. Huang, L. Zhao, S. Kvatinsky, S. Yu, X. Liu, J. Kang, Y. Nishi, and H.-S. P. Wong, "Analysis and Predication on Resistive Random Access Memory (RRAM) 1S1R Array," *Proceedings of the 2015 International Memory Workshop*, pp. 1-4, May 2015.

[8] S. Sheu, P. Chiang, W. Lin, H. Lee, P. Chen, Y. Chen, T. Wu, F. T. Chen, K. Su, M. Kao, K. Cheng, and M. Tsai, "A 5ns Fast Write Multi-Level Non-Volatile 1 K Bits RRAM Memory with Advance Write Scheme," *Proceedings of the Symposium on VLSI Circuits*, pp.82-83, June 2009.

[9] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic Design within Memristive Memories Using Memristor-Aided loGIC (MAGIC)," *IEEE Transactions on Nanotechnology*, Vol. 15, No. 6, pp. 1-16, July 2016.

[10] E. Linn, R. Rosezin, S. Tappertzhofen, U. Bttger, and R. Waser, "Beyond von Neumann Logic Operations in Passive Crossbar Arrays Alongside Memory Operations," *Nanotechnology*, Vol. 23, pp. 305205:1–6, July 2012.

[11] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaacobi, and S. Kvatinsky, "Logic Operation in Memory Using a Memristive Akers Array," *Microelectronics Journal*, Vol. 45, No. 11, pp. 1429-1437, November 2014.

[12] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive Switches Enable 'Stateful' Logic Operations via Material Implication," *Nature*, Vol. 464, pp. 873-876, April 2010.

[13] E. Lehtonen, J. H. Poikonen, and M. Laiho, "Two Memristors Suffice to Compute All Boolean Functions," *Electronics Letters*, Vol. 46, No. 3, pp. 239-240, February 2010.

[14] S. Shin, K. Kim, and S.-M. Kang, "Reconfigurable Stateful NOR Gate for Large-Scale Logic-Array Integrations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 58, No. 7, pp. 442-446, July 2011.

[15] E. Lehtonen, J. H. Poikonen, J. Tissari, M. Laiho, and L. Koskinen, "Recursive Algorithms in Memristive Logic Arrays," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, Vol. 5, No. 2, pp. 279-292, June 2015.

[16] R. Ben Hur, N. Talati, and S. Kvatinsky "Algorithmic Considerations in Memristive Memory Processing Units (MPU)," *Proceedings of the International Workshop on Cellular Nanoscale Networks and their Applications*, August 2016.

[17] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling," *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 14–23, December 2009.